Strong Consistency for Shared Objects in Pervasive Grids

Luiz Angelo Steffenel, Manuele Kirsch Pinheiro

Strong Consistency for Shared Objects in Pervasive Grids

Luiz Angelo Steffenel
Département de Mathématiques et Informatique
Université de Reims Champagne-Ardenne
BP 1039, F-51687 Reims Cedex 2, France
email: Luiz-Angelo.Steffenel@univ-reims.fr

Manuele Kirsch-Pinheiro
Centre de Recherche en Informatique
Université Paris I - Panthéon-Sorbonne
90 rue de Tolbiac, F-75013 Paris, France
email: Manuele.Kirsch-Pinheiro@univ-paris1.fr

***Abstract*—Recent advances in communication technology enable the emergence of a new generation of applications that integrates mobile devices with classical high performance systems as part of a common computing environment. In such environments, keeping the coherence of shared data (distributed objects, for example) represents a real challenge as communications are strongly influenced by the performance and the reliability of mobile devices (laptops, PDAs and cellular telephones) and wireless networks (WiFi, Bluetooth). Indeed, data incoherence may arise due to message losses or node volatility, which blocks the algorithms used to synchronize these data. In this paper, we analyze the main challenges concerning the manipulation of shared distributed objects in a pervasive environment. We demonstrate how a membership service can be enhanced to tolerate temporary disconnections and message losses without blocking, while reducing the number of exchanged message.**

I. Introduction

The widespread availability of mobile devices (PDAs, smartphones, etc.) and of wireless networks, such as WiFi and GSM, has boosted mobile and pervasive computing. The term pervasive computing refers to the seamless integration of devices into the users' everyday life [1]. This term represents an emerging trend towards environments composed by numerous computing devices that are frequently mobile or embedded in the environment and that are connected to a network infrastructure composed of a wired core and wireless edges [2].

When considering pervasive environments, one should consider heterogeneous environments composed of fixed and mobile devices interconnected by a mix of standard infrastructures (fixed networks) and wireless networks (Fig. 1). These mobile nodes are equipped with standard and/or wireless communication interfaces that allow them to move at will, as well as allowing them to connect over a fixed structure (as in the case of laptop computers). In such an environment, nodes that are located at the boundaries of the wireless coverage zone may be out of reach from time to time. Also, mobile devices that have low power capacities may disconnect themselves to save battery power.

Different applications built on the top of mobile devices can benefit from our membership algorithm, especially those that need to ensure a coherent view of a data set such as a multiplayer game in an ad-hoc network [3], the collaborative edition of a document [4] or even distributed computing in a pervasive grid or P2P environment [5]. In addition, our attention was especially drawn to the aspects of component deployment and state transfer among mobile devices. Indeed, several authors have been studying dynamic component deployment on pervasive systems [6], [7], [8], and we are especially interested in the case of preventive deployment, in which components are pro-actively deployed in order to keep the system responsiveness.

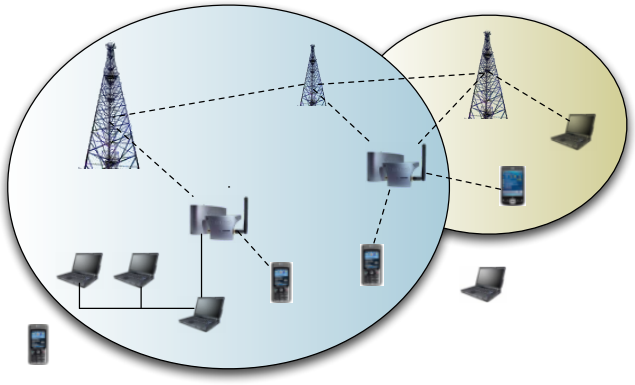

Fig. 1. Topology with mobile devices

For illustration purposes, let us consider a mobile device such as a PDA or a cell phone, whose battery discharges. In such case, pervasive systems may decide to deploy components from this device to other device before that the battery reaches a critical level. If the system waits until the last moment to perform this deployment and the state transfer necessary to keep running, chances are that the transfer fails before completing the deployment. To avoid this, a mobile device may contact neighbor devices before reaching a critical state and manage replicas of the current application (including the concerned components and their corresponding states) on those devices.

Therefore, distributed systems running on the top of pervasive environments (we call them pervasive systems) have to cope with problems such as node volatility and network coverage. Unfortunately, we cannot rely on traditional dis-

tributed systems as they do not target these problems: not only traditional algorithms assume rare failures but also they mainly focus on the occurrence of nodes failures [9].

To cope with the problems that arise from a dynamic and volatile environment, we propose a solution based on a Group Membership Service specifically tailored for pervasive systems. Our proposal is constructed around the concept that most disconnections are due to the network coverage problems and not to a node failure. Therefore, we advocate that suspected processes should no be immediately removed from the current membership, at least as long as they do not block the application. Hence, according to the failure situation, different algorithms may apply: an algorithm that removes suspected processes and install a new membership, or a lazy algorithm that simply reorganizes the processes to prevent (or to delay) blocking situations. Using such approach, we minimize the number of membership changes, who depend on expensive operations such as the Consensus [10]. We believe that this two-level structure considerably improves the system liveness when temporary disconnections of processes take place. Hence, to illustrate its mechanism, we examine a distributed computing scenario where data consistency over shared objects is required in spite of mobile devices temporarily disconnections.

This paper is organized as follows: In Section II we recall basic properties from group communication and membership, and establish the system model. Section III introduces the problem of Group Membership in the context of pervasive systems, illustrating the drawbacks of traditional techniques. For instance, Section IV propose a two-level membership algorithm that tolerates temporary disconnections in a dynamic pervasive system, while V presents an efficient approach for implementing Atomic Broadcast in such system. Finally, Section VI presents the conclusions of this work and future directions.

II. Group Communication Definitions

When working in a distributed system, group communication operations and group membership are important tools to simplify the coordination among distributed processes. More specifically, a group membership service considers the problem of managing the successive memberships of a group of processes (usually called *views*), keeping them coherent under some properties. Basically, a group membership requires three primitives, namely *join* (by which a process ask to join the group), *leave* (by which a process as to leave the group) and *install* (by which a new view is approved). A process can also be excluded from a view when it is suspected to have crashed. In this paper we consider only the primary-partition membership service [11], where we attempt to keep a single view of the current group.

To help managing the group membership, View Synchronous Communication (or VSC, for short) [11] allows processes to broadcast messages with certain guarantees. Let V-BROADCAST^v denote the primitive by which a message is broadcast by a process in view v, and V-DELIVER^v the primitive that delivers a message to a process in view v. VSC is defined by the following properties (we consider here non-Uniform properties):

Validity - If a ***correct*** process[1] V-BROADCAST^v a message m, then some ***correct*** process eventually V-DELIVER m to the application (in view v or in a subsequent view).

Termination - If a process V-BROADCAST$^v(m)$, then eventually (1) every process in view v V-DELIVER$^v(m)$ or (2) every correct process in v installs a new view.

View Synchrony - If a process p belongs to two consecutive views v and v' and V-DELIVER$^v(m)$ in view v, then every process q in $v \cap v'$ that installs v' also V-DELIVER$^v(m)$ before installing v'.

Integrity - For any message m, every ***correct*** process p V-DELIVER(m) at most once and only if (1) m was previously V-BROADCAST by *sender(m)* and (2) p is a process in the set Π.

Sending View Delivery[2] - If a process p V-BROADCAST(m) in a view v, then every correct process ought to V-DELIVER(m) in the same view v.

The Group Membership problem can be solved by reduction to Consensus [10], [12]. Informally, the Agreement property[3] helps a view change algorithm to define the same view among processes. As the scope of Group Membership also considers messages exchanged among the processes, it is used by several works to define operations such as Atomic Broadcast or Reliable Broadcast on message sets [11], [13].

With respect to communications, we consider *fair-lossy* channels that provide reliable communication using *unreliable* channels by ensuring that a message m is retransmitted until its successful reception (signaled by an *ack*, for example).

III. Group Membership on Pervasive System

As stated in Section II, a membership view change gathers all correct processes in a new view v_{i+1}. Furthermore, to ensure that all processes in the new view are coherent, these processes share all queued messages and deliver them before installing the view v_{i+1}. A view change depends therefore on the agreement among processes.

Traditional membership algorithms assume that devices are connected by reliable networks and that disconnections are rare, and therefore are not designed to support group management in pervasive environments [14]. Indeed, most membership specifications strongly rely on the Consensus operation, which requires not only a majority of correct nodes but also that they remain connected as long as the agreement has not been reached. The intrinsic volatility from a pervasive environment may lead a simple Consensus to be delayed for several rounds if nodes connect and disconnect regularly (even if at any given time t there is a majority of connected nodes).

To minimize the dependency of membership view changes on the Consensus operations, especially in the case of a

[1] A process is called *correct* only if it does not crash during the entire execution, although even a correct process can be incorrectly suspected of crashing

[2] Some specifications consider a weaker property called "Same View Delivery" instead of "Sending View Delivery" [11]

[3] Agreement : no two processes decide differently [10]

pervasive environment that is prone to frequent disconnections, we advocate the use of a two-level membership view change, where consensus is used only as the last resort. In this scheme, nodes suspected are initially put into "quarantine" but not removed from the group, allowing suspected nodes to overcome temporary disconnections.

IV. A Two-level View Change Mechanism

The first step to efficiently handle temporary disconnections in a pervasive system is to specify a group membership service with mechanisms to tolerate these disconnections while keeping a good reactivity on the case of process failures. As suggested in [15], we consider that each view change level defines different types of views: *regular views* (or simply *views*) are similar to the views of View Synchronous Communication, while *intermediate views* (or *i-views*) are installed between regular views.

If regular views are denoted by $v_0, v_1, ..., v_i$, the i-views between v_i and v_{i+1} are denoted as $v_i^0, v_i^1, \ldots, v_i^j, \ldots, v_i^{last}$. The intermediate view v_i^0 is equal to v_i and the last intermediate view v_i^{last} is equal to v_{i+1}. One important point is that the membership of all intermediate views $v_i^0, v_i^1, \ldots, v_i^{last-1}$ is the same as the membership of v_i, that is, they only differ in the order that processes are listed in the view. For example, $v_i = v_i^0 = \{p, q, r\}$, $v_i^1 = \{q, r, p\}$, etc. As a result, i-view changes can be optimized to interfere a minimum with the system operation. In this paper we propose to redefine a group using the concept of "quarantine": a view is therefore composed by two subgroups, {**"active","suspected"**}, where i-view changes simply move suspected processes to the corresponding subgroup.

The specification of this two-level membership is identical with respects to the properties from Section II, except for the Sending View Delivery property that becomes:

Sending View Delivery - If a process *p* V-Broadcast(*m*) in a view *v*, then every correct process ought to V-Deliver(*m*) in the same regular view v (*i-view* changes could have occurred between).

Such a two-steps membership presents several advantages for pervasive systems, as it allows the system to adapt to temporary disconnections without inducing a regular view change. As we use two different failure detectors, we can fine tuning each one to reflect the pervasive environment: i-views can be triggered by failure detectors with aggressive timeouts or *ad-hoc* suspicions (e.g.: a process that does not succeeds sending a message to other process), while regular views are triggered by failure detectors with conservative timeouts.

Please note that we rely on non-Uniform properties mainly because they can allow a less costly implementation in pervasive systems. To ensure strong completeness, however, we must use *program-controlled crash* [16]: if a message is broadcasted in view *v* and all correct processes should deliver the messages broadcasted in the same view *v*, VSC forces suspected processes to crash, ensuring the *Sending View Delivery* property. Our mechanism minimizes the situations where program-controlled crash may apply as *program-controlled crash* are triggered only when *i-views* are no more able to manage processes in a view.

Even reducing the probability of Regular View changes, several i-view changes may occur before reaching stability. In the next section, we present a lightweight algorithm for i-view changes that does not rely on Consensus, reducing its impact on pervasive systems.

A. Optimizing i-view changes

From the previous sections, we can define an algorithm for the V-Broadcast and Regular View Change, as presented in Algorithm 1.

Algorithm 1 V-Broadcast and Regular View Change algorithm

V-Broadcast(m) executed by p_k:
send (i, m) to all process in v_i

Upon reception of (i, m) by p_k while in view v_i
V-Deliver(m)
add m to $unstable_k$

Upon suspicion of some process in v_i by a conservative failure detector
R-Broadcast (view-change, i) /* R-Broadcast is defined in [13]*/

Upon R-Deliver (view-change, i) by p_k for the first time

1. send $unstable_k$ to all
2. $\forall p \in v_i$, wait until receive $unstable_l$ from p_l or p_l suspected
3. let $initial_k$ be the tuple (Π_k, $Msgs_k$) s.t.
   - Π_k is the set ($p_i \cup$ processes that sent their $unstable_l$)
   - $Msgs_k$ is the union of the $unstable_k$ sets received
4. execute Consensus among v_i processes, with $initial_k$ as the initial value
5. let (Π, *Msg*) be the Consensus decision
6. V-Deliver all messages in Msg not yet V-Delivered
7. if $p_k \in \Pi$, then "install" Π as the next view v_{i+1}
   else suicide

Here, messages sent with V-Broadcast are kept in the queue $unstable_k$ until they become stable. Once a process receives the unstable queue from all processes that are not suspected, it can compute $Msgs_k$, the union of all received $unstable$. It also can suggest a new view based in the set of processes that answered the *view-change* message. As processes agree both on the new view and on the set of unstable messages, all processes that acknowledge this decision have the same set of messages and therefore these messages are ready to be delivered. Please note that suspected nodes excluded from the view are forced to suicide (Algorithm 1, line 7).

In the case of i-views changes, we don't need to ensure the Sending View Delivery property. Therefore, we concentrate on a lightweight algorithm for i-view changes that only deals with processes suspicions, as presented in Algorithm 2.

Algorithm 2 Optimized i-view changes

Upon suspicion of some process q in v_i^j by an aggressive failure detector
R-Broadcast (i-view, i, j, q)

Upon R-Deliver (i-view, i, j, q) by p_k for the first time

1. If suspected(q) then *Broadcast (i-view,i, j,ack)*, else *Broadcast (i-view,i, j,nack)*
2. $\forall p$, wait until a majority of votes for ack or nack is reached
3. if majority(ack)
   - move q to the "suspected" subgroup in the set Π
   - install Π it as the next i-view v_i^{j+1}

This optimized i-view algorithm no longer forces processes to manipulate lists of messages at each i-view change, which makes i-views even lighter than the regular view change algorithm. By reducing the overhead on i-view changes, we reduce the impact of wrong suspicions due to aggressive failure detectors. Similarly, the fact that i-views do not force a process to suicide reduces the overhead induced by the membership service.

B. Proof of correctness

In this section, we sketch the proofs of correctness for the VSC properties in our algorithms. Consider V-BROADCAST(m) and the current view v_i:

Lemma 1: Sending View Delivery is satisfied.

Proof: m can only be V-DELIVERED in view v_i, this is ensured by tagging each message with the current view number. ■

Lemma 2: View Synchrony is satisfied.

Proof: m can only be V-DELIVERED in view v_i, and (*i*) either all members of v_i eventually V-DELIVER(m) or (*ii*) a new view v_{i+1} is defined and if one process V-DELIVERS m before installing v_{i+1}, then every process that installs v_{i+1} has V-DELIVERED m before installing the new view. ■

Lemma 3: Termination is satisfied.

Proof: If not all process in v_i V-DELIVERED m, then some process has crashed and if we assume a $\diamondsuit\mathcal{W}$ failure detector, the crashed process is eventually suspected. So R-BROADCAST(*view-change, i*) is executed and a new view is eventually installed. Let p_i be a process that is in the new view and has V-DELIVERED m before installing the new view. We show that each process that installs the new view has V-DELIVERED m:

- Case 1: p_i has detected the stability of m before sending $unstable_i$ to all. By definition of stability, all processes in v_i have V-DELIVERED m.
- Case 2: m was not stable at p_i. Let p_k be the process whose initial value is the decision (Π, Msg). By item 7 we have $p_i \in \Pi$ and by item 3 p_k has received the $unstable_i$ set from p_i. So $m \in Msg$ and by item 6 every process has V-DELIVERED m before installing the new view.

■

We also observe that i-views do not interfere with the VSC properties. On i-view changes, each suspected process is handled individually and a majority of votes is required to decide on the suspicion. Our algorithm is more resilient than a simple failure detector because we require a majority of commitments to move a process to the "suspected" subgroup; contrarily to Consensus, our algorithm does not force all processes to install the same view v_i^{j+1}. Indeed, we assume that a "majority test" is sufficiently enough to define intermediate views. In the case processes install different i-views and one of these i-views blocks the application (keeping a crashed process in the *active* set), eventually a Regular View change will be triggered, solving the problem.

C. Performance issues

Let us assume that i-view changes are triggered by a failure detector with a small timeout (e.g. 1s) and regular view changes are triggered by a failure detector with a conservative timeout (e.g. 50s). In the case of a temporary disconnection (or a failure), i-view changes allow us to react much faster than a standard VSC with a timeout of 10s, improving the liveness property. If finally a failure suspicion is confirmed, regular view changes cost in average 50s (worse than VSC), but we reduce the probability of incorrectly excluding processes, minimizing the cost of *program-controlled crash*.

To understand the advantages of both regular and i-view compared to a standard membership algorithm, we must understand that the crash of a process interferes with the group only if the group depends on that process (waiting for a message or trying to send a message to it). As long as i-view changes are able to prevent blocking situations, we avoid expensive regular view changes, which is especially interesting in the case of pervasive networks.

V. STUDY CASE

To illustrate our approach, please consider a distributed computing environment where processes must share an object. In a previous work [5], developed a purely decentralized peer to peer middleware for grid computing called CONFIIT (Computation Over Network with Finite number of Independent and Irregular Tasks). CONFIIT was designed to address the problem of efficiently deploy scientific problems that can be parallelized as independent tasks. Among such problems there are classical combinatorial problems such as N-Queens, Langford and car-sequencing.

Because tasks under CONFIIT are independent, almost no synchronization is required among the processes. Indeed, the single element that nodes need to synchronize is the list of completed tasks, which is ensured by a token passing. This computational model clearly impacts on the fault-tolerance aspects of CONFIIT, as processes that disconnect cause almost no harm (the worst case being the regeneration of the token).

In the scenario that we propose, however, processes not only share more complex objects but require data consistency in order to respect a task-dependency graph. It is clear that such kind of application will suffer if deployed in a pervasive environment, as the disconnection of a process may block the progress in a graph path. Indeed, we need to ensure non-blocking data consistency in the shared objects even when mobile devices disconnect temporarily.

Another interesting scenario could be represented by a mobile application that undergo a preventive deployment. In this scenario, an application running on a machine with a low battery level may decide to migrate to other devices in a transparent way. Here, devices must keep consistency on a shared distributed object even if new events (data) arrive from different sources during the migration. Devices that enter sleep mode for a few seconds (until being plugged to the AC adapter) may disrupt the migration process if no attention is made.

A. Ensuring consistency

One of the best known operations to ensure data consistency in a distributed environment is the Atomic Broadcast operation. The Atomic Broadcast (sometimes called Total Order broadcast [17]) is a group communication primitive that ensures that processes in a distributed system deliver messages to the application respecting the same order. This global delivery order is essential when implementing services that require coherence between processes such as distributed databases or collaborative edition. This problem can be defined by four properties (Validity, Agreement, Integrity and Total Order) [18]. Validity and Integrity are basic properties, while Agreement and Total Order definitions are presented below:

Agreement - If a ***correct*** process delivers a message m, then all ***correct*** processes in Π eventually deliver m.

Total Order - If ***correct*** processes p and q both deliver messages m and m', then p delivers m before m' if and only if q delivers m before m'.

Several techniques can be used to ensure these properties [18], such as Fixed [19] or Moving Sequencer [20], Privilege Based [21], Communication History [22] and Destination Agreement [10]. Hybrid approaches also exist, such as FSR [23] that is based on the fixed-sequencer strategy but uses a token ring to ensure fairness among the nodes. Other recent works on Atomic Broadcast try to relax some constraints in order to improve performance. Indeed, [24], [25], [26] assume that the network often provides spontaneous total order, requiring special procedures only when this assumption does not hold.

B. Implementing Atomic Broadcast

With a few exceptions, most Atomic Broadcast algorithms rely on local area networks where disconnections are rare and communication times can be easily bounded. Unfortunately, frequent disconnections may force an Atomic Broadcast algorithm based on Consensus [10] to execute several rounds before a majority of processes agreed on a message order. In such a scenario, message delivery will be blocked until the gathering of a stable quorum.

To better handle temporary disconnections, we focus on the *moving sequencer* strategy [20], [27]. which presents the performance of a sequencer-based implementation while preventing a single point of failure by rotating the role of sequencer among the nodes. As the *moving sequencer* strategy does not rely on consensus, it can perform faster in a pervasive environment than consensus-based techniques.

The moving sequencer strategy can be easily implemented using a token-passing algorithm (see Fig. 2). Due to the lack of space we will not detail the algorithm, but the principle is as follows: when a process q wants to broadcast a message m, it sends m to all other processes. Upon receiving m, processes store it into a receive queue. When the current token holder p has a message in its receive queue, it uses the sequence number to timestamp the first message in the queue and broadcasts that sequence number together with the token. In a non-Uniform algorithm, a process can then deliver m when it has (1) received m, (2) received m's sequence number, and (3) delivered every message with a smaller sequence number.

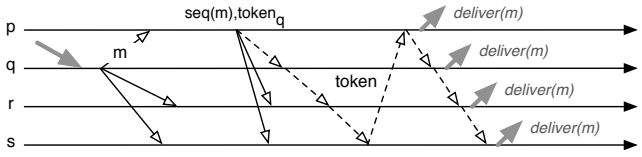

Fig. 2. The *token-passing* mechanism with Uniform delivery

With respect to fault tolerance, a token-passing mechanism must be aware of two different situations: (*i*) the crash of a process and (*ii*) the loss of a token. In the first case, the token passing is blocked because the "next sequencer" has failed and cannot receive the token; in the second case, the current token holder crashes before the next sequencer is able to accept the token (for example, the next sequencer lacks some previous messages). It is clear that solving these two situations requires different measures. In the first case, it is enough to redefine the virtual ring, removing the crashed process. In the second situation, processes must not only agree a new virtual ring but also choose a process to restart the token passing.

Therefore, to adapt this token passing mechanism to a volatile pervasive system, we suggest integrating the token passing mechanism with the membership group description presented in the previous section. Using that two-fold structure (and the V-BROADCAST operation), a process that blocks the token passing is moved to the "suspected" subgroup. Then we can restart the token passing only among processes in the "active" subgroup. As only active processes participate in the message sequencing, we minimize the probability of blocking the token (Fig. 3). As "suspected" processes still belong to the group view, they can receive all messages sent to the group and even submit new messages (a token holder can assign sequence numbers to messages not of its own).

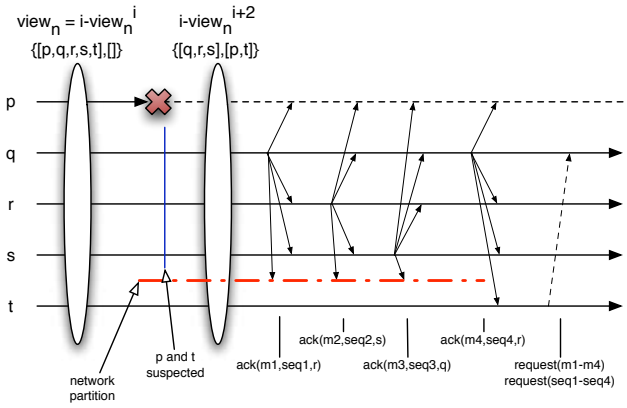

Fig. 3. I-views and suspected members

When a process is moved to the "suspected" set, we cannot make any assumption about its state (we do not know if a suspected process has really failed or not). For this reason, it is

important that "active" processes ensure total order properties. This allows correct processes in the "suspected" set to be kept updated and request lost messages, while waiting to be reintegrated to the "active members" in a future view change. Consequently, this membership mechanism can cope with short disconnections commonly found in wireless networks: as the token is passed only among stable nodes, we drastically reduce the events that trigger a new Regular View.

However, suspected processes cannot be reintegrated in all cases. Processes in the *suspected* subgroup that reconnects after a long absence may be unable to acquire missing messages (which could have been delivered and removed from the buffers after ensuring message stability). In this case, theses processes must "suicide" and reconnect with a different ID. When a new process *joins* the group, it triggers a Regular View change, becoming from that moment coherent with the other processes in the view.

VI. Conclusions

In this paper, we addressed the problem of ensuring strong data consistency for share objects in the context of pervasive systems. Traditional algorithms are not fit for these environments as they cannot handle the nodes volatility efficiently. We propose a group membership solution that can operate in environments subjected to frequent disconnections. In order to ensure a smooth operation in spite of the volatility of the resources, we employ a two-level membership organization to minimize the problems generated by wrong failure suspicions, thus reducing the need for view changes and expensive Consensus. Our efforts now concentrate on two subjects: conducting experiments in a pervasive P2P environment, evaluating the impact of both nodes heterogeneity and volatility on the algorithms behavior and developing deployment/migration mechanisms for mobile devices using the proposed algorithms.

References

[1] M. Baldauf, S. Dustdar, and F. Rosenberg, "A survey on context-aware systems," *International Journal of Ad Hoc and Ubiquitous Computing*, vol. 2, no. 4, pp. 263–277, 2007.

[2] T. Moran and P. Dourish, "Introduction to this special issue on context-aware computing," *Human-Computer Interaction*, vol. 16, no. 2-3, pp. 87–95, 2001.

[3] R. Spanek, P. Kovar, and P. Pirkl, "The bluegame project: Ad-hoc multilayer mobile game with social dimension," in *3rd International Conference on emerging Networking EXperiments and Technologies (CoNEXT'07)*. ACM Press, December 2002.

[4] H. Skaf-Molli, C.-L. Ignat, C. Rahhal, and P. Molli, "New work modes for collaborative writing," in *International Conference on Enterprise Information Systems and Web Technologies (EISWT-07)*, Orlando, Florida, USA, Jul. 2007.

[5] O. Flauzac, M. Krajecki, and J. Fugère, "CONFIIT: a middleware for peer to peer computing," in *The 2003 International Conference on Computational Science and its Applications (ICCSA 2003)*, ser. Lecture Notes in Computer Science, M. Gravilova, C. Tan, and P. L'Ecuyer, Eds. Montréal, Québec: Springer-Verlag, Jun. 2003, vol. 2669 (III), pp. 69–78.

[6] Y. Vandewoude and Y. Berbers, "Component state mapping for runtime evolution," in *Proceedings of the 2005 International Conference on Programming Languages and Compilers*, Las Vegas, Nevada, USA, June 2005, pp. 230–236.

[7] P. Rigole, T. Clerckx, Y. Berbers, and K. Coninx, "Task-driven automated component deployment for ambient intelligence environments," *Pervasive and Mobile Computing*, vol. 3, no. 3, pp. 276–299, June 2007.

[8] Y. Vanrompay, Y. Berbers, and P. Rigole, "Learning-based coordination of distributed component deployment," in *1st International DisCoTec Workshop on Context-aware Adaptation Mechanisms for Pervasive and Ubiquitous Services*, ser. Electronic Communications of the EASST. EASST, June 2008, to appear.

[9] K. P. Birman, *Reliable Distributed Systems: Technologies, Web Services and Applications*. Springer, March 2005.

[10] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM*, vol. 43, no. 2, pp. 225–267, 1996.

[11] G. Chockler, I. Keidar, and R. Vitenberg, "Group communication specifications: a comprehensive study," *ACM Computing Surveys*, vol. 33, no. 4, pp. 427–469, 2001.

[12] L. Lamport, "The part-time parliament," *ACM Transactions in Computer Systems*, vol. 16, no. 2, pp. 133–169, 1998.

[13] A. Schiper, "Dynamic group communication," *Distributed Computing*, vol. 18, no. 5, pp. 359–374, 2006.

[14] D. Bottazzi, A. Corradi, and R. Montanari, "Agape: a location-aware group membership middleware for pervasive computing environments," in *ISCC*, 2003, pp. 1185–1192.

[15] B. Charron-Bost, X. Défago, and A. Schiper, "Broadcasting messages in fault-tolerant distributed systems: the benefit of handling input-triggered and output-triggered suspicions differently," in *Proceedings of the 21th International Symposium on Reliable Distributed Systems*, 2002.

[16] T. D. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost, "On the impossibility of group membership," in *Proc. of the 15th ACM Symposium on Principles of Distributed Computing*, May 1996, pp. 322–330.

[17] V. Hadzilacos and S. Toueg, *Fault-tolerant broadcasts and related problems*, 2nd ed. ACM Press Books, Addison-Wesley, 1993, ch. 5, pp. 97–146.

[18] X. Défago, A. Schiper, and P. Urbán, "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Computing Surveys*, vol. 36, no. 4, pp. 372–421, December 2004.

[19] K. Birman, A. Schiper, and P. Stephenson, "Lightweight causal and atomic group multicast," *ACM Transactions on Computer Systems*, vol. 9, no. 3, pp. 272–314, 1991.

[20] J.-M. Chang and N. Maxemchuk, "Reliable broadcast protocols," *ACM Trans. on Computer Systems*, vol. 2, no. 3, pp. 251–273, 1984.

[21] R. Ekwall, A. Schiper, and P. Urbán, "Token-based atomic broadcast using unreliable failure detectors," in *Proceedings of the 23rd Symposium on Reliable Distributed Systems (SRDS 2004)*, Florianópolis, Brazil, Oct. 2004.

[22] D. Dolev, S. Kramer, and D. Malki, "Early delivery totally ordered broadcast in asynchronous environments," in *Proceedings of the 23nd Annual International Symposium on Fault-Tolerant Computing*, Toulouse, France, June 1993, pp. 544–553.

[23] R. Guerraoui, R. Levy, B. Pochon, and V. Quéma, "High throughput uniform total order broadcast protocol for cluster environments," in *Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN)*, 2006.

[24] P. Vicente and L. Rodrigues, "An indulgent uniform total order algorithm with optimistic delivery," in *Proceedings of the 21st IEEE International Symposium on Reliable Distributed Systems (SRDS'02)*. IEEE Computer Society Press, 2002, pp. 92–101.

[25] B. Kemme, F. Pedone, G. Alonso, A. Schiper, and M. Wiesmann, "Using optimistic atomic broadcast in transaction processing systems," *IEEE Transactions in Knowledge Data Engineering*, vol. 4, no. 15, pp. 1018–1032, 2003.

[26] A. Souza, J. Pereira, F. Moura, and R. Oliveira, "Optimistic total order in wide area networks," in *Proceedings of the 21st IEEE International Symposium on Reliable Distributed Systems (SRDS'02)*. IEEE Computer Society Press, 2002, pp. 190–199.

[27] B. Whetten, S. Kaplan, and T. Montgomery, *A High Performance Totally Ordered Multicast Protocol*. Springer-Verlag, Berlin Heidelberg New York, 1995.